Bell Laboratories Cover Sheet for Technical Memorandum

Title- **The C Language Calling Sequence**

Date- **September 26, 1977**

TM- **77-1273-15**
77-1274-13

Other Keywords-

Author	Location	Extension	Charging Case- **39199**
S. C. Johnson	**MH 2C-559**	**3968**	Filing Case- **39199-11**
D. M. Ritchie	**MH 2C-517**	**3770**	
M. E. Lesk	**MH 2C-572**	**6377**	

ABSTRACT

One of the most critical issues in the implementation of a C language system is the design of the calling sequence. Poor decisions in this area can cause substantial time and space penalties for all C programs; most seriously, it is extremely painful to change calling conventions after a C system is in use, since this typically requires finding and recompiling all existing C programs.

To make good choices for calling conventions, one must weigh carefully the register and addressing structure of the machine, and any relevant operating system conventions. It is also usualiy desirable to arrange the calling conventions so that the stack size need not be predefined, and so that postmortem debugging stack traces can easily be produced.

These objectives can rarely be simultaneously attained; in general, compromises must be made. This document attempts to set forth the major issues, and discusses experience with the PDP-11, GCOS, and IBM C environments. An Appendix discusses the calling sequence for the Interdata 8/32 C compiler.

Pages Text 11	Other 3	Total 14	M. J. Melchner
No. Figures 0	No. Tables 2	No. Refs. 1	

SEE REVERSE SIDE FOR DISTRIBUTION LIST

BELL TELEPHONE LABORATORIES, INC.

DISTRIBUTION
(REFER GEI 13.9-3)

TM-77-1273-15
TM-77-1274-13

COMPLETE MEMORANDUM TO

CORRESPONDENCE FILES

OFFICIAL FILE COPY
PLUS ONE COPY FOR
EACH ADDITIONAL FILING
CASE REFERENCED

DATE FILE COPY
(FORM E-1328)

10 REFERENCE COPIES

<AHO,ALFRED V
<BAKER,BRENDA S
<BECKER,RICHARD A
RIBEN,IRMA R
HOLVIE,RICHARD H
<BOYCE,W M
BROWN,E L
BROWN,EDRIN L
<BROWN,W STANLEY
CERMAK,I A
CHANEY,F W
CHEN,STEPHEN
CLAYTON,D P
CROME,L L
DICKMAN,BERNARD N
FABRICIUS,WAYNE N
<FELDMAN,STUART I
<FLEISCHER,H I
FOUGHT,R T
<FRASER,A G
FRESHMAN,K G
FRYDMAN,UBSZULA D
GABEY,MICHAEL E
GEYLING,F T
<GOLDSTEIN,A JAY
<GRAHAM,R L
GRAYSON,G F,JR
<GUTHERY,S B
+HANNAY,N B
HOLTZMAN,JACK M
<JOHNSON,STEPHEN C
<KEESE,W M
<KERNIGHAN,BRIAN W
LIEN,Y EDMUND
LIU,HUEI-CHI RICHARD
<LINDSEY,GOTTFRIED W R
<MARANZANO,J F
+MC DONALD,H S
+MC GILL,R
MCILROY,M DOUGLAS
MILLER,ALAN H
MILLER,GERALD L
MILTON,DONN R
MORGAN,DENNIS J
<MORGAN,SAMUEL P
MORRIS,ROBERT
<OSSANNA,J F,JR
PETERSON,RALPH W
<PILLA,M A

COMPLETE MEMORANDUM TO

SPINSON,ELLIOT N
+PRIM,R C
<RALEIGH,T M
<RIDDLE,GUY G
<ROBERTS,CHARLES S
RODRIGUEZ,ERNESTO J
ROWEN,GEORGE
ROWLAND,BRUCE R
SEKINO,W T
SETHI,RAVI
<SLOANE,N J A
SPIRES,R J
STROHECKER,CARY A
SUEN,LAI-CHEENG
TERRY,M E
THOMPSON,K
WATSON,D S
<WEINBERGER,PETER J
<YAMIN,ELAINE E
68 NAMES

COVER SHEET ONLY TO

CORRESPONDENCE FILES

4 COPIES PLUS ONE
COPY FOR EACH FILING
CASE

ACKERMAN,A FRANK
AHRENS,RAINER H
ALCALAY,D
AMRON,IRVING
ANDERSON,FREDERICK L
ANDERSON,KATHRYN J
APPELBAUM,MATTHEW A
ARMSTRONG,D R
ARNDT,DENNIS L
ARNOLD,GEORGE W
ARNOLD,S L
ARNOLD,THOMAS F
ATAL,BISHNU S
BACH,MARIECE J
BALDINI,J J
BARCLAY,DAVID K
BASELL,RICHARD J
BAUER,ANDREW E
BAUER,HELEN A
BAUER,S C
BELLOWS,J S
BERGLAND,G D
BERNSTEIN,DANIELLE R
BERNSTEIN,L
BEYER,JEAN-DAVID
BIANCHI,M H
BICKFORD,NEIL B
BILLINGTON,MARJORIE J
BILOWUS,B M
BIRCHALL,R H

COVER SHEET ONLY TO

BISHOP,VERONICA L
BLAZIER,S D
BLINN,J C
BLOSSER,PATRICK A
BLUE,JAMES L
BLUMER,THOMAS P
BLUM,MARION
BOCKUS,ROBERT J
BODEN,F J
BONANNI,L E
BOURNE,STEPHEN R
BOWYER,L RAY
BOYER,PHYLLIS J
BOYLE,GERALD C
BOYLE,W S
BRADLEY,E H
BRAUNE,DAVID P
BROSS,JEFFREY D
BROWN,W R
BULPER,A F
BULLEY,E M
BURG,F M
BURNETTE,W A
<BURNETT,DAVID S
BURROW,STEVEN J
BURROWS,THOMAS A
BUTLETT,DARRELL L
BYRNE,EDMUND R
CAMPBELL,JERRY H
<CANADAY,BUDD H
CARBOSA,WAYNE M
CASELLA,CHARLES R
CASPERS,BARBARA E
CAVINESS,JOHN D
CHAMBERS,B C
<CHAMBERS,J W
CHANDRA,RAKESH
CHAPPELL,S G
CHEN,D
<CHERRY,LORINDA L
CHESSON,GREGORY L
CHILDS,CAROLYN
CHODROW,M M
CHRISTENSEN,S W
CHRIST,C W,JR
CHU,PAUL H N
COBEN,ROBERT M
<COLE,LOUIS M
CONDON,J H
CONNERS,RONALD E
COOPER,JOSHUA E
COOPER,MICHAEL H
COPP,DAVID H
CORNELL,B G
COSTELLO,PETER E
CRAGUN,DONALD W
CRISTOFOR,EUGENE
CUTLER,C CHAPIN
DAVIS,R DREW
DE FAZIO,M J
DE GRAAF,D A

COVER SHEET ONLY TO

DENSMORE,WAYNE
DEUTSCH,DAVID N
<DIMMICK,JAMES O
DINEEN,THOMAS J
DOLOTTA,T A
DONNELLY,MARGARET M
DOWDEN,DOUGLAS C
DOWDEN,IRIS S
DRAKE,LILLIAN
DROSEIKIS,FREDERICK C
D'ANDREA,LOUISE A
DUFFY,F P
DWYER,T J
EIGEN,D J
<EITELBACH,DAVID L
ELLIOTT,G L
ELLIOTT,RUBY J
ELY,T C
EPLEY,ROBERT V
ESCOLAR,CARLOS
ESSERMAN,ALAN R
ESTES,VIRGINIA DANIELLE
ESTOCK,RICHARD G
FABISCH,M P
FEDER,J
<FELS,ALLEN M
FELTON,WILLIAM A
FERIDUN,K K
FINUCANE,J J
FISCHER,HERBERT R
<FISHER,T S
FISHMAN,DANIEL H
FLANDENA,R
FOMI,K T
FORSBURG,ALLEN S
FORTNEY,V J
FOUNTOUKIDIS,A
FOWLER,H EUGENE
FOWLKES,EDWARD B
FOX,PHYLLIS A
FOY,J C
FRANCIS,SAMUEL H
FRANK,AMALIE J
FRANK,RUDOLPH J
FREIDENREICH,S
FROST,H BONNELL
FROCHTMAN,GARRY
FULTON,A W
GALLANT,R J
GARST,BLAINE,JR
<GATES,G W
GAY,FRANCIS A
GEARY,M J
GEORGEN,MICHAEL R
GEPNER,JAMES R
GIBB,KENNETH R
<GILLETTE,DEAN
GILL,R J
GIMPEL,J F
GITHENS,J A
GLASSER,ALAN L

COVER SHEET ONLY TO

GLICK,F G
<GNANADESIKAN,R
GOGUEN,N H
GOLAREE,RUTH T
GOLDBERG,HAROLD JEFFREY
<GORMAN,J E
GORTON,D R
GREENBAUM,HOWARD J
GROSS,ARTHUR G
GRZESZEKI,MAUREEN E
GUIDI,PIER V
HAFER,E H
HAGGERTY,JOSEPH P
HAIGHT,R C
HALE,A L
HALL,ANDREW D,JR
HALL,MILTON S,JR
HALL,WILLIAM G
HAMILTON,LINDA L
<HAMILTON,PATRICIA A
<HARKNESS,CAROL J
HARRISON,H
HARRIS,MARION O
HARUTA,K
HAUSE,A DICKSON
<HAWKINS,DONALD T
HAWKINS,RICHARD R
HAYDEN,DONALD F,JR
HENDERSON,CECELIA E
HERGENHAN,C B
HEROLD,J W
HOEHN,MARIE J
HOLTMAN,JAMES P
HOMREDEL,J H
HOU,RAYMOND
HOWARD,PHYLLIS A
HOYT,WILLIAM F
HO,DON T
HO,JENNY
HUNNICUTT,C F
IEBLEY,W H
IPPOLITI,O D
IRVINE,M M
<IVIE,EVAN L
JACKOWSKI,D J
JACKSON,H M,2ND
JACOBS,H S
JAMES,J W
JARVIS,JOHN F
JEFFREY,H JOEL
<JENSEN,PAUL D
JESSUP,E R
JOHNSON,D S
JOHNSON,DAVID O
JOHNSON,R L
<JUDICE,C N
KACHURA,J J
<KAISER,J F
KAMIN,SCOTT J
KANE,J RICHARD
KANODIA,RAJENDRA K

467 TOTAL

+ NAMED BY AUTHOR > CITED AS REFERENCE < REQUESTED BY READER (NAMES WITHOUT PREFIX WERE SELECTED USING THE AUTHOR'S SUBJECT OR ORGANIZATIONAL SPECIFICATION AS GIVEN BELOW)

MERCURY SPECIFICATION.....

COMPLETE MEMO TO:
127-SUP

COTCCO = COMPUTING THEORY, COMPILING

COTES SHEET TO:
12-DIR 13-DIR 127

COPLGP = COMPUTING/PROGRAMMING LANGUAGES/GENERAL PURPOSE

HO CORRESPONDENCE FILES
HO 5C101

TM-77-1273-15
TOTAL PAGES 14

PLEASE SEND A COMPLETE

( ) MICROFICHE COPY ( ) PAPER COPY

TO THE ADDRESS SHOWN ON THE OTHER SIDE.

TO GET A COMPLETE COPY:

1. BE SURE YOUR CORRECT ADDRESS IS GIVEN ON THE OTHER SIDE.
2. FOLD THIS SHEET IN HALF WITH THIS SIDE OUT AND STAPLE.
3. CIRCLE THE ADDRESS AT RIGHT. USE NO ENVELOPE.
4. INDICATE WHETHER MICROFICHE OR PAPER AS DESIRED.

Bell Laboratories

Subject: **The C Language Calling Sequence**
Case- 39199 -- File- 39199-11

date: **September 26, 1977**

from: **S. C. Johnson**
D. M. Ritchie
M. E. Lesk

TM: **77-1273-15**
77-1274-13

MEMORANDUM FOR FILE

Introduction

One of the most critical issues in the implementation of a C language system is the design of the calling sequence. Poor decisions in this area can cause substantial time and space penalties for all C programs; most seriously, it is extremely painful to change calling conventions after a C system is in use, since this typically requires finding and recompiling all existing C programs.

To make good choices for calling conventions, one must weigh carefully the register and addressing structure of the machine, and any relevant operating system conventions. It is also usually desirable to arrange the calling conventions so that the stack size need not be predefined, and so that postmortem debugging stack traces can easily be produced. The other languages available on the computer, and the costs and advantages of compatibility with them, should also be evaluated.

These objectives can rarely be simultaneously attained; in general, compromises must be made. This document attempts to set forth the major issues, and discusses experience with the PDP-11, GCOS, and IBM C environments.

The Basic Issues

Functions in C must permit recursion; moreover, C programs can take the addresses of local variables and arguments, and pass these addresses down to other functions; this more or less rules out any form of "copy-in/copy-out" implementation of recursion, where the local variables are stored in static cells and only saved when a function is reentered recursively. Thus, most C environments will have a stack, and local variables and arguments must be saved on this stack. The amount of stack space needed can be data dependent, and difficult to estimate in advance. Thus, it is desirable to permit the stack to grow up to the available limits of addressable memory. Failing this, stack overflow should be clearly signaled and easily fixed.

The amount of stack space used by typical programs can be greatly reduced by moving automatic arrays off the stack. Since the semantics of C generally permit an array to be replaced by a pointer to a suitable block of storage, on-stack arrays can be transformed into pointers with initialization calls to allocation routines. Since stack limits, if they must be checked, must be checked on every subroutine entry, a considerable time saving is possible if stack usage can be kept small enough that overflow need not be checked.

do check on GP

To further complicate the situation, C programs (in particular, the function *printf*) may be called with a variable number of arguments. In general, the *calling* program knows how many arguments are passed, while the *called* program does not, *a priori*. On the other hand, the *called* program knows how many automatic variables and temporary locations will be needed, while the *calling* program does not, *a priori*. Thus, establishing the called program's environment includes doing the necessary storage allocation and making the arguments available to the called program. It is also necessary to reverse this process upon return from the called program, freeing the allocated storage, restoring the environment of the caller, and making the value returned from the function (if any) available to the caller.

Splitting the Work

Parts of the job of calling and returning can be done either by the caller or the called function. This section discusses some of the tradeoffs.

As mentioned above, some of the status of the caller must be saved at the call and restored upon return. This status includes the function return address, information which allows arguments and local variables to be addressed (e.g. the stack pointer), and the values of register variables, if any. This information can typically be saved either by the caller or the called program. If the calling program saves the information, it knows which registers contain register variables, and does not need to save registers which do not contain register variables. On the other hand, the called program does not need to save registers which it will not change (although this makes the compiler's job a bit harder).

Space considerations may play a role in the decision. If the status is saved by the called program, the save/restore code is generated once per *function;* if the saving is done by the caller, the code is generated once per *call* of the function. Since all functions are presumably called at least once, and many functions are called more than once, considerable code space can be saved by putting the burden of the status saving on the called function.

There is considerable advantage to having the return sequence common and shared by all routines. Besides the obvious saving in space, there is a more subtle advantage: if the code is common, this implies that returns can be executed without knowledge of what routine was being returned from. This is exactly what is needed to permit an easy implementation of stack traces, or postmortem dumps, even after wild transfers or faults; this has considerable advantage in the construction of debugging environments. If the caller's status information is to be stored by the called program, it must then be in a known position in the called program's environment, for all programs.

To summarize this section, we suggest that the caller's status be saved by the called program, in a known place in the called program's environment; this permits shared common code for returns, and straightforward implementation of stack traces. Some C systems use shared code for saves also, although the cost of remembering what was called must be balanced against the benefit of sharing the save sequence.

Passing the Arguments

As suggested above, the calling program must assume primary responsibility for passing the arguments, since it alone knows how many there are to be. Although it is possible to contemplate C environments where this is not true, it is usually understood that the arguments to a function can be treated as an array; a pointer to the first argument can be used to access the second and successive arguments. This implies that the arguments must be stored in increasing order somewhere in memory. Moreover, the arguments should begin at a known position in the called program's environment, in order that they be efficiently accessed by the called program.

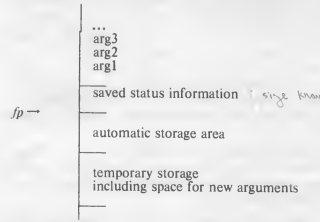
The establishment of the arguments to the called program can be accomplished by several methods. The calling program might simply establish an array with the arguments, and pass a pointer to this array which is maintained in a base register in the called program environment.

This is effective, but leads to two base registers being needed (one for arguments, one for automatics), with a consequent increase in the amount of status information which must be saved. Another strategy is to pass such a pointer, and the size of the argument region, and let the called routine copy the arguments into the same area as its automatic variables. This saves a base register, but costs a copy. Since 98% of functions have fewer than 4 arguments,[1] there may be enough registers to pass the first several arguments in registers, and resort to a copy operation only with functions which have a large number of arguments. Since many machines have relatively inexpensive save operations which will save multiple registers, these registers might be copied quite cheaply into the automatic variable region of the environment. With a bit of luck, this copying process might also be able to save most of the status of the calling program with the same multiple register save. Finally, it may happen that the stack allocation algorithm permits the caller to know where the called program's stack frame will be; in this case, it may be most efficient to cause the calling program to store the arguments directly into the area where the called program will expect to see them. In practice, this requires that the stack be in a contiguous area of memory. Nevertheless, this strategy is important enough to rate a section to itself.

Perhaps the Best Strategy

If one has enough control of the operating system environment, it may be possible to place the C stack in an area which can be grown contiguously in one direction or the other. For the moment, it will be assumed that the stack is growing backwards (from high to low memory), and that it is possible to index both positively and negatively off base registers; other cases will be discussed later.

We suggest a layout for the stack frame which involves the use of a single base register, or *stack frame pointer (fp)*. The stack frame looks like:



where high memory addresses are at the top of the page. The amount of status information saved is assumed known to the called routine; if this is *x* bytes (or whatever the addressing unit is...), then the arguments can be addressed as x(*fp*), x+4(*fp*), etc. (where the constants reflect the size of the arguments). The automatic variables can be accessed as -4(*fp*), -8(*fp*), etc... The saved status information is always addressable at 0(*fp*), and thus is in the same place in every routine, facilitating debugging.

To call a function, the calling program first stores the arguments in increasing order at the very end of its temporary area. It then transfers control to the called program, putting the address of the arguments into a special register in order to pass this information to the called routine. The called routine then saves the status of the caller, including the old value of *fp*,

1. H. Gajewska and S. C. Johnson, *Some Statistics on the Usage of the C Language*, TM 75-1273-13.

and then uses the passed argument pointer to establish the new value of *fp*. To return, the function value being returned is placed into a special register, and then the status of the caller, including the old *fp* value, is reestablished. The entire process is quite simple, and can be implemented to run reasonably quickly.

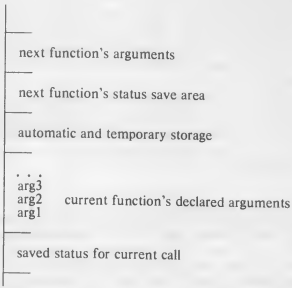
This organization is basically that used in the original C implementation on the PDP-11. There is additional hardware on the PDP-11 which can be effectively used with this organization. In particular, there is a hardware stack pointer (*sp*) register, and the instruction set permits words to be moved onto the stack more rapidly than a general move from one arbitrary location to another. If the *sp* register is kept pointing to the lower end of the current stack frame, it can be used to put arguments, the return address, and the saved register variables onto the stack relatively quickly.

We have said nothing about checking for stack overflow. On the PDP-11, the stack overflow check is done as part of the usual memory protection hardware; if a reference to a stack location violates the memory protection, the operating system attempts to allocate more memory and extend the stack segment downwards. If more memory is available, the offending instruction is restarted. If all memory is exhausted, the process is aborted. This scheme has the advantage that the overhead takes place only when the stack is extended. The hardware needed to support this is rather sophisticated, however, since it must be possible to restart instructions which fault in the middle of their execution, and to extend protected memory regions downward in memory. If this hardware is lacking, the test for explicit stack overflow is only about two additional instructions per call.

Perhaps the most unnatural thing about this organization is the stack growing backwards. One thing this does is permit the data area of the process to begin at low addresses, and be extended, by contiguous allocations, upwards in memory, while the stack area can begin at high addresses and grow downwards. It does require hardware, however, which will support protected regions growing downward; this hardware seems to be nearly unique to the PDP-11. On machines with a larger word size, there may be plenty of address space, allowing a segment or more to be devoted to the stack. It is thus worthwhile considering the difficulties of permitting the stack to grow upwards. Clearly, the above organization could simply be turned on its head, and the result would still be an efficient and effective organization. The only incompatibility would be that arguments would now form an array which would run backwards in memory! If this array were turned around, to run forwards in memory, the problem is that the called program does not know how many arguments there are, and cannot figure out the address of the first one without knowing this information.

One way out of this difficulty is to force the called routine to declare at least as many arguments as it expects to receive. This allows us to set aside enough space for the declared arguments, and any additional arguments are simply lost. This leads to a somewhat ugly incompatibility, but does not seriously affect the power of the language.

Notice that it must still be possible to index both positively and negatively off *fp*. Unfortunately, a number of machines (notably the IBM 360/370 series) take all offsets from base registers to be positive; this means that additional work is needed to preserve the good features of the above organization. To begin, the stack must almost certainly run forward in memory, since addressing locations below the current frame pointer is very painful. Since the automatic variables and the arguments must both be put above the place pointed to by the *fp*, in order to efficiently address them, calling routine should store the arguments where the called routine expects them; however, the calling routine does not know the size of the automatic region for the called routine, so the arguments must lie below the automatic region. If the saved status is to be put in a known position in the stack frame, it must come before both the arguments and the automatics, i.e., first. The final organization is as follows (high memory addresses are at the top):



The call will involve establishing the arguments of the called routine, setting a register to point to the beginning of the next function's status save area, and transferring control. The called function will then save the status, and, if necessary, check to ensure that there is enough stack space available to go into execution. It then simply sets the *fp* and begins. To return, the old status, including the return address, is loaded from the status save area, and a return branch is done; the *fp* is automatically restored as part of the restoration of the status.

To summarize, this section has described in a fair degree of detail a C stack frame organization which provides reasonable efficiency, supports debugging, and requires only a single base register. Effective use of this requires a considerable degree of control over the process environment, however. Other considerations will be touched on in the next several sections.

Alignments

The PDP-11 has very few alignment restrictions for the kinds of data with which the machine deals. Most operations must be on word (two byte) boundaries. In particular, this implies that the stack frame pointer must always be word aligned.

On other machines, the alignment restrictions can be much stricter. For example, on the Interdata 8/32, integers must be placed on 4 byte boundaries, and doubles on 8 byte boundaries. This latter restriction forces the frame pointer, and, by extension, the frame size, to be a multiple of 8 bytes. Moreover, certain other addresses, such as the base of the status save area, the base of the arguments, and the base of the new status save area and new argument area, must all have the same alignment restrictions. This is not difficult to achieve in the portable compiler, but it does need to be done.

The *nargs* function

Occasionally, when a function is designed to work with a variable number of arguments, it is desirable to know how many arguments were actually passed. The function *nargs* was written to address this problem; when called from within a function, it returns the number of arguments in the current call to that function. Unfortunately, the PDP-11 hardware makes implementation of *nargs* when I and D space are separated very expensive. Thus most programs and library functions in the UNIX* operating system are not dependent on knowing the exact number of arguments with which they were called (notice that *printf* does not need *nargs*, since it has a format string with which to pick apart the given arguments).

* UNIX is a Trademark/Service Mark of the Bell System

There are also some philosophical problems with *nargs*. When arguments have different sizes (as **long**s, **int**s, and **double**s do on the PDP-11), then it may be possible to determine the total size of all the arguments passed, but much more difficult to determine the exact number. If there are alignment restrictions (for example, that doubles or longs must be on even word boundaries), the effect of this may be to cause even the size of the argument space to be rounded up to the next 8 byte boundary.

The "ideal" calling sequence described above does not admit any easy way of finding out how many actual arguments were used, or even how much argument space was used. Clearly, this information could be placed into a strategic register as part of the call, but this would add to the call overhead. It appears that in most systems the increased power would not be worth significant increased time and space in the call.

Interrupts

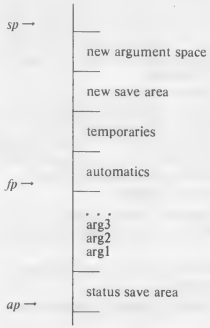
In the UNIX system, interrupts (in the system) and signals (in user processes) are turned into function calls. These functions require stack space, and it is convenient to use the same stack as the ordinary routines. This implies that it must be possible to recognize the last location used in the current stack frame, to know where to begin the stack frame for the interrupt routine. A register pointing to the end of the stack frame (call it *sp*) would typically require one additional instruction to set up the register upon entry to the call, and would be an extra register of status information to be saved.

The exact order in which things must be done in the call is quite subtle in order that an interrupt not use a piece of the stack area that is still needed by the current function. For example, having called a new function, if the status save area is to be placed in the new stack frame, *sp* must be incremented before the status is saved, in order that an interrupt not use the space before *sp* is incremented. Of course, incrementing *sp* destroys the old *sp* value, which was one of the things which was to be saved. In general, to escape these problems, it is necessary at least to copy *sp* to a safe place during the prolog; in the worst case additional stores or other computation may need to be done.

An alternative is to allocate the stack for interrupts from some other area of memory (e.g., the top of the data area...). This is aesthetically ugly, but could be an improvement over throwing away another register on some machines. Alternatively, one could attempt to ensure that the signal routine would leave extra space for a possible status save in progress; this is quite dangerous, but could probably be done if carefully thought out. A similar issue arises in swapping routines out (*sp* marks the end of the area to swap) and in reentrant routines (such as some interrupt handlers), and must be handled similarly in all cases.

Argument Pointers

There is increasing interest in coroutine environments for C programs. In such environments, it is inappropriate to speak of 'the stack', since there may be many stacks, and, in particular, the assumption that the arguments to a function are adjacent to the stack space for that function may be violated. In this case, it is natural to indicate the locations of the arguments by means of an argument pointer register, *ap*. For debugging reasons, it is desirable to be able to access the saved registers in a fixed location. Notice that storing the register status information at a fixed location off of *ap* gives us a solution to the two-step problem discussed under interrupt handling. It then becomes the caller's job to set aside a save area for the status, but the called program still does the save. The stack frame layout looks like:



This arrangement also allows the called program to access its arguments without any need to know how many there will be, or to declare a maximum number.

In the ordinary case, where the functions are used in a stack-like manner, the overhead can be quite modest. In particular, another register is needed to hold *ap*, and this register must be saved and restored across calls; before a call, the new argument pointer must be established (usually with some form of 'load address' instruction), and then, in the called routine, the new argument pointer must be copied into *ap*. Care must be taken that *ap* satisfy the strictest alignment requirements of any arguments that may be passed. These operations can be done quite quickly on many machines; this approach seems quite attractive on machines with a reasonable number of registers, such as the Interdata 8/32 (See the Appendix).

Functions Returning Structures

In the near future, it is expected that C will gain the ability to pass structures and unions by value as arguments to functions, and to return structures and unions as values from functions, as part of the general implementation of structure assignment. This has some interesting effects on the implementation of the C calling sequence.

Structures will be passed into a routine by simply copying the value onto the stack; as long as the caller and called function agree on the amount of space and the alignment, the details are relatively unimportant. It is function values returned from functions which are the real problem. The difficulty here is to find the appropriate place to store the returned value. Since structure values do not fit into machine registers, the caller and called programs must agree on a place where the called routine will leave the value where the caller can get it and copy it. If the called function allocated some space, and returned a pointer to this space to the caller, there would be several difficulties. If the space were allocated on the stack, it would be 'unprotected' after the return and before the value was copied; a signal, interrupt, or swap at this time could be a disaster. If the space were allocated as a static area, the situation would be little better; the value would be more protected, but the function would still not be reentrant, and could not be safely called from within a signal or interrupt processor.

The solution seems to be to allocate the space for the value in the caller, and communicate the space to the called function, which takes responsibility for copying the value before returning. This communication could be done by passing an optional argument or setting another register on call, but it is easiest perhaps to simply arrange that the returned value

overlay the argument area in the caller's stack frame. This requires that the caller leave enough space for the returned value when this is more than the space needed for the arguments. The caller must also be careful not to reuse the argument area to call another function before the value is copied or otherwise dealt with. The called function, in its turn, must be careful when copying the return value so that if a function returns one of its arguments, the value is properly copied. In effect, this forces the copy to be done in a forward direction.

It is tempting to suggest that functions which return small structures, say one or two words, should return those values in registers, as they would for **long** or **int** values. This is attractive, but, without saying it cannot be done, it is harder than it at first appears. The central problem seems to be that code sequences arise as a result of this optimization which do not arise at any other time. Since this usage is expected to be rather slight, there is a large increase in complexity and the potential for bugs to gain only a small advantage in speed and space.

As examples of some of the strange things which might arise, consider:

1. A structure consisting of three characters, on a 32 bit machine. Returning this in a register gives us the unenviable task of assigning three characters in a register to three successive bytes in memory, of arbitrary alignment.
2. A structure consisting of an **int** and a **float.** Should the structure return the first word in a general purpose register, and the second in a floating register?
3. A union of a **float** and an **int.** Should the result come back in a general purpose register or a floating register?

There are numerous other cases, involving assignment of these strangely placed structures, passing them to other functions, and extracting members from them. All of these special cases would have to be enumerated, made to work, and tested. After repeated attempts to do something of this nature for the Interdata, we finally gave up on all optimization.

In passing, note that structure valued arguments to functions makes *nargs* even less well defined than it was previously.

The Real World

The calling sequences described above are used on the PDP-11 and the AP3; the proposed C environment for the Interdata 8/32 also uses a similar organization. On both GCOS and IBM, the C compiler generates rather different calls, for similar reasons. On both systems, it is in general not possible to get more stack space contiguously, and on both systems there were existing system calling conventions which seemed to offer considerable advantages for compatibility.

On GCOS, the C calling sequence is essentially the same as the calling sequence for FORTRAN; control is transferred, and the addresses of arguments are in memory following the call. The called routine obtains enough space for its stack frame, and then copies the arguments into the proper place on this stack frame. Efficiency is helped by the powerful indirect addressing capabilities of the Honeywell hardware, which allow references to automatics, arguments, and constants to be compiled directly into argument lists, frequently without executing any code at all. This scheme also permits the stack to be discontiguous, so additional space can be obtained from the operating system if needed. There are difficulties with variable length argument lists, however; some of these are inherent in the FORTRAN call (on GCOS, FORTRAN programs will frequently abort if functions are called with fewer arguments than they expect, even if the missing arguments are never referred to). The problem is that the called routine tries to copy in arguments which do not exist; the result can be an indirection through a word which does not specify a legal address, leading to a fault. This was annoying at first, and soon became intolerable. It was fixed by adding four instructions to each call which check on the number of arguments passed, and copy only those arguments which were actually present. The other problem with this copying is that the called routine does not always know the size of the argument, at least with floats and doubles. It must thus rely on the declared size, which makes it impossible to write a general *printf* function easily. This problem was met by providing a

machine-dependent program which would make available a list of addresses of the arguments to the called program; this enabled *printf* and its variants to be written.

On the IBM/370, we chose to be compatible with the Indian Hill Bliss calling sequence. Since C and Bliss are similar languages, we planned to make use of the Bliss I/O library. In retrospect, this was probably not too wise, as the Bliss library has been of no use in the OS and TSO environments, and some of the Bliss conventions make interfacing with standard OS calls somewhat more difficult.

Were the IBM calling sequence to be reimplemented, it seems that the large address space and the availability of virtual memory on most IBM systems would argue strongly for an organization similar to that of the last section.

If any general conclusion can be drawn, it is that the problem of implementing a C calling sequence when the operating system is not ideal depends on many issues rather far removed from C itself; if the language is to be used to implement system code, many UNIX conventions can be ignored, but speed is probably important. If the aim is to obtain many UNIX utilities with relatively little effort, it may be worth spending a bit more time in the call in order to keep the fine points compatible with the PDP-11.

An Eye Towards UNIX

Since the UNIX operating system is written in C, and has been successfully moved to the Interdata 8/32, it is reasonable to ask what effects UNIX considerations might have on the C calling sequence design.

There is some code in the UNIX system itself which depends on the calling sequence: interrupt handlers, machine language routines (such as system call interface routines) which are called from C, etc. all must have some knowledge of the calling sequence conventions, at a fairly low level. Moreover, portions of the system such as the code to swap programs out and in, and to grow programs which have run out of stack space, must know whether the stack is running forwards or backwards.

A very complicated area involves the machine language functions *savu, retu,* and *aretu.* Without going into great technical detail, these functions allow a process to save its stack information in such a way that it can later resume execution. The functions are at the heart of the *fork* function of UNIX, and, in a real sense, are what makes multiprogramming work. They depend critically on the calling sequence, and have proved to be a real trouble spot on the Interdata. On the plus side, these routines are called in only a few places in the depths of the system, so that more or less arbitrary things can be done to the calls in order to make the functions work. The purpose of this section is simply to indicate a difficult area which must be addressed and solved before UNIX can be moved.

Timings

Since calling sequences represent a significant fraction of the total processor time used by C programs, it is important to evaluate the effects on runtime of various alternatives. When the IBM calling sequence was being studied, measurements were taken to estimate the costs of the different possible calling sequences. There was a running IBM compiler, using a particular calling sequence; the question was whether to change to a BLISS, ALGOL, or FORTRAN compatible call.

To obtain measurements, a program was run at Holmdel and the CPU time charged was noted. The same program was then run to find out how many subroutine calls are made, using the histogrammer available with GCOS C. The subroutine calls were then timed by running a program at Holmdel which just made calls, but did nothing else. To inconvenience the cache memory, ten different null subroutines were called in a loop.

Since timings seem to vary by about 0.2 seconds at random, all programs were run several times and the results averaged. To run a null program which does nothing much took 0.33 seconds of CPU time. 100000 subroutine calls took 1.35 seconds. Thus, each call of the old

IBM C calling sequence averaged about 10.2 microseconds.

The first program measured was the UNIX file comparison utility *diff*, which took 2.23 seconds on a sample job that involved 21247 subroutine calls. Hence, we estimate that *diff* spends 11% of its time doing subroutine calls. Two trivial routines were also timed; one indicated 17% and the other 28% of total time performing calls.

The measurements of the compiler itself were much more important. To compile one short program involving 8163 subroutine calls took 0.82 seconds of CPU time, implying about 17% of total time in calls. Compiling a larger program (the c00.c file of the compiler) involved 144230 subroutine calls and took 6.10 seconds; this indicates 25% of total time in calls.

The various calling sequences considered are shown in the following table. Times are taken from the "Functional Characteristics" manual for the 370/165; although out of date, only relative timings matter.

Possible Calling Conventions

Language	Cost of call (microseconds) per call	per argument
C (old)	3.79+4s+.32g	.64+2s
Algol 68	4.99+2s	.48+2s
Bliss	2.75+2s+.32g	.48+2s
Fortran	5.43+.32g+12s	1.12+5s

s = cost of a storage reference
g = number of general registers saved

To assign numbers to the various expressions, the number of general registered stored was averaged to 10 (g=10), except that the old C call always stored 15. The cost of a memory reference (s) is assumed to be 1.3 microseconds times 10% references out of cache, which is 0.13 microseconds. The number of arguments per subroutine call is assumed to average 1.5.

The next table shows the cost of the various calling sequences.

Table 1

Time of Subroutine Interface (μseconds)

Style of call	Per Argument	Prolog, Epilog	Total Time
C now:	.90	9.43	10.78
C improved:	.74	7.51	8.62
Algol 68:	.74	5.25	6.36
Bliss:	.74	6.21	7.32
Fortran:	1.77	10.19	12.84

The calculated time of 10.78 microseconds per call agrees well with the observation of 10.2 microseconds per call; among possible errors are the use of Model 165 timings on a Model 168, and better cache efficiencies. Considering the old C calling sequence with some minor improvements, Table 2 gives the time taken to perform a job that takes one second with that calling sequence, given the fraction of the time spent executing calling sequences.

Table 2

Cost of different calls

% Time in calls	Algol 68	Bliss	Fortran
0	1.00	1.00	1.00
5	0.99	0.99	1.02
10	0.97	0.98	1.05
15	0.96	0.98	1.07
20	0.95	0.97	1.10
25	0.93	0.96	1.12
30	0.92	0.95	1.15
35	0.91	0.95	1.17
40	0.90	0.94	1.20
45	0.88	0.93	1.22
50	0.87	0.92	1.24

Since the numbers above indicate that an appropriate number for fraction of time in calls is 20%, changing to the Bliss calling sequence would speed programs up by roughly 3%, while changing to the Fortran compatible calling sequence would slow programs down 10%. Independently, we estimated that it would save perhaps 8% to pass the first argument to every routine in a register.

Having made these measurements, we then chose the BLISS call as fast and politically desirable.

Summary

We have discussed the major issues involved in the construction of a C calling sequence, and have given a stack frame organization which we expect to be the basis for future implementations of C on new machines. An appendix discusses the calling sequence for the Interdata 8/32 C compiler.

S. C. Johnson

D. M. Ritchie

M. E. Lesk

MH-1273/1274-SCJ/DMR/MEL-unix

Att.
Appendix

Appendix: the Interdata 8/32

The Interdata 8/32 is a machine rather similar to the IBM 370 series machines, with the significant exception that most instructions permit a "long address" address mode capable of addressing any element in memory without using a base register. Thus, it is not necessary to have many base registers dedicated to establishing addressability, although base register references to data are faster and smaller than the equivalent longer forms.

The offsets from base registers are unsigned 14 bit numbers, and the memory protection does not allow segments to grow backwards; thus, the stack grows forwards in memory.

To establish a baseline, first consider the fastest calling sequence that is marginally acceptable; this is one with only a single frame pointer register, which does not support interrupts or debugging stack traces. Throughout the following, let *V* be the number of register variables which are saved in the calling sequence. In the fastest calling sequence, a call looks like

```
la      nfp,offset(fp)
bal     link,subroutine
```

where *fp* is the frame pointer, *nfp* the new frame pointer, and *link* the subroutine linkage register. The called subroutine need merely do

```
stm     xx,0(nfp)
lr      fp,nfp
```

where **xx** is the first register to be saved. (The instruction saves registers starting at its argument through register 15). The return sequence is

```
lm      xx,0(fp)
br      0(link)
```

The registers are assigned so that the **lm** and **stm** cause the loading and storing of the *V* register variables, the *link* register, and the old *fp*. An approximate timing is

$$13.85 + 1.55V$$

microseconds. Thus, with 3 register variables saved, the cost would be 18.50 microseconds.

As pointed out in the body of the text, the handling of interrupts makes it attractive to have a pointer to the end of the current stack frame. This register must be saved and restored (1.55 microseconds) and reestablished for each call (1.12 microseconds); moreover, in order that the registers not be saved beyond *sp* it must be copied (.4 microseconds). This gives a cost of

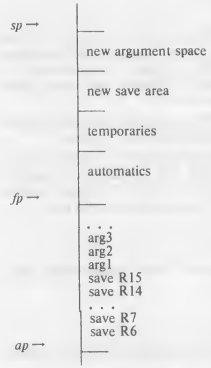
$$16.92 + 1.55V$$

With three register variables, this gives a time of 21.57 microseconds. The unpleasant parts of this calling organization are the need to declare more arguments than would be passed to a function, and the costlier implementation of coroutines. If another register is dedicated to hold an argument pointer, this register would have to be saved and restored across calls (1.55) and, in addition, the frame pointer would have to be established (.4) in each new routine. On the other hand, *sp* would no longer have to be copied in the called routine (saving .4 microseconds). Thus, this most ornate calling sequence would take

$$18.47 + 1.55V$$

microseconds. With three register variables, this time is 23.12 microseconds; this is roughly 25% longer than the stripped down call, and less than 10% slower than the call with *sp* alone. Moreover, the Interdata, with 16 registers, is not short of registers. Thus, this calling sequence was adopted.

The stack frame organization looks like:



To make a call, first the arguments are placed at the end of the current stack frame. Then the caller would execute

```
la      nap,offset(fp)
bal     link,subroutine
```

The subroutine would do

```
stm     xx,yy(nap)
lr      ap,nap
lr      fp,sp
la      sp,offset(fp)
```

Upon return, the subroutine would execute:

```
lm      xx,yy(ap)
br      0(link)
```

Here, xx and yy are functions of *V* which are chosen to make the registers saved always lie at the end of the save area.

The following organization permits this calling sequence within the constraints of the **lm** and **stm** instructions:

0	scratch register
1	base register to the data area (never changed)
2	scratch register: used for function returns
3	scratch register
4	scratch register
5	scratch register: holds *nap* on calls
6—11	register variables
12	*ap*
13	scratch register: holds *link* on calls
14	*sp*
15	*fp*

This organization allows up to 6 register variables.

The Interdata machine has writable microstore; instructions to do the save and return sequence could be easily microcoded, and would run faster, although the exact numbers await empirical study.

Measurements and Empiricism

For several reasons, the calling sequence implemented in the Interdata at this writing is somewhat slower than indicated here. For one thing, a (hopefully temporary) hardware condition forces us to check explicitly for stack overflow; it is enough for us to generate the instruction

```
l       r0,0(sp)
```

at the end of the called program's prolog, in order to generate a fault if there was stack overflow. Another empirical problem is that, when the code is unoptimized, certain numbers (for example, the xx and offset values above) are known only at the end of the function. In the absence of optimization, the assembler may use longer forms for these instructions than are necessary. Finally, for various reasons relating to optimization all return sequences have an extra branch in them; some have two. These will be removed as the optimizer gets debugged and shaken down, but they are there now. Thus, the measured time to do a call of a null function is longer than expected; it closely fits the function

$$24. + 2.3\ V$$

microseconds, or almost 25% slower than calculated.